NPS-MA-07-002

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

Masking a Compact AES S-box

by

David Canright

07 August 2007

Approved for public release; distribution is unlimited

Prepared for: Naval Postgraduate School
Monterey, CA 93943

This report was prepared for Naval Postgraduate School

Reproduction of all or part of this report is authorized.

This report was prepared by:

____
David Canright
Associate Professor

Reviewed by:

____
Clyde L. Scandrett
Department of Applied Mathematics

Released by:

____
Dan C. Boger
Interim Associate Provost and
Dean of Research

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** August 2007	**3. REPORT TYPE AND DATES COVERED** Technical Report

4. TITLE AND SUBTITLE: Title (Mix case letters) Masking a Compact AES S-box	**5. FUNDING NUMBERS**
6. AUTHOR(S) David Canright	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER** **NPS-MA-07-002**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES
The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

When the Advanced Encryption Standard (AES) is implemented in hardware or software, it may be vulnerable to "side-channel attacks" such as differential power analysis. One countermeasure against such attacks is adding a random mask to the data; this randomizes the statistics of the calculation at the cost of computing "mask corrections." The single nonlinear step in each round of the AES algorithm is called the "S-box," which involves the greatest computational cost in a round (to find the inverse in the Galois field), as well as the greatest cost for mask corrections. Oswald et al.[9] showed how the "tower field" representation allows maintaining an additive mask throughout the Galois inverse calculation. This work combines that masking approach with the compact S-box of Canright, to give a masked S-box that requires minimal circuitry, and hence the chip area.

14. SUBJECT TERMS AES, CRYPTOGRAPHY, MASKING, SIDE-CHANNEL ATTACKS	**15. NUMBER OF PAGES** 25
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Masking a Compact AES S-box

D. Canright
Applied Mathematics Dept., Code MA/Ca
Naval Postgraduate School
Monterey, CA 93943

7 August 2007

Abstract

When the Advanced Encryption Standard (AES) is implemented in hardware or software, it may be vulnerable to "side-channel attacks" such as differential power analysis. One countermeasure against such attacks is adding a random mask to the data; this randomizes the statistics of the calculation at the cost of computing "mask corrections." The single nonlinear step in each round of the AES algorithm is called the "S-box," which involves the greatest computational cost in a round (to find the inverse in the Galois field), as well as the greatest cost for mask corrections. Oswald et al.[9] showed how the "tower field" representation allows maintaining an additive mask throughout the Galois inverse calculation. This work combines that masking approach with the compact S-box of Canright, to give a masked S-box that requires minimal circuitry, and hence the chip area.

1 Introduction

The Advanced Encryption Standard (AES) was specified in 2001 by the National Institute of Standards and Technology [8], to provide a standard algorithm for secure encryption, intended not only for U.S. government documents, but also for electronic commerce.

Many different implementations of AES have appeared, to satisfy the varying criteria of different applications. Some approaches seek to maximize throughput, e.g., [6], [15] and [5]; others minimize power consumption, e.g., [7]; and yet others minimize circuitry, e.g., [12], [13], [16], and [3]. For the latter goal, Rijmen[11] suggested using subfield arithmetic in the crucial step of computing an inverse in the Galois Field of 256 elements. This idea was further extended by Satoh et al.[13], using sub-subfields (the "tower field" representation of Paar[10]), along with other innovative optimizations, which resulted in the smallest AES circuit at that point. The architecture of Satoh was refined somewhat by Canright[2], mainly through carefully chosen normal bases, resulting in the most compact S-box to date.

No attacks have yet been found on the AES algorithm itself that are more effective than exhaustive key search ("brute force"), although research continues, for example, on algebraic attacks. But hardware implementations of cryptograpy, e.g. in smart cards, may be vulnerable to "side-channel attacks" such as differential power analysis, that use statistical analysis of side effects like power consumption, electromagnetic radiation, etc., to deduce information about the secret key.

One countermeasure against side-channel attacks is masking the data during calculation through adding or multiplying by random values. All the steps in a round of AES are affine, except for the Galois field inversion substep of the S-box (*SubBytes*) step. For the other steps, calculation of the mask correction is linear, so an additive mask is most convenient. Some have suggested switching to a multiplicative mask for the Galois inverse step (e.g., [1]), but one inescapable weakness is that a zero data byte is unmasked by multiplication [4].

Applying the "tower field" representation, inversion in $GF(2^8)$ involves several multiplications and one inversion in the subfield $GF(2^4)$, which in turn involves multiplications and inversion in $GF(2^2)$. In the sub-subfield $GF(2^2)$, inversion is identical to squaring, and so is *linear* (over $GF(2)$). Oswald *et al.* applied this idea to additive masking of the Galois inverse, and showed how to compute the mask correction for the tower field approach. Many of the correction terms involve multiplication in subfields, and Oswald *et al.* showed how some of these multiplications can be eliminated through clever re-use of parts of the input mask for the output.

The present work incorporates this masking approach into the compact S-box of Canright[2]. Applying the same optimizations used there for the unmasked S-box, to the mask correction terms here, results a compact *masked* S-box.

2 Algebraic description

The AES algorithm has been described thoroughly and frequently elsewhere[8]; here we give the barest outline before concentrating on the S-box. It is a symmetric block (16 bytes) cipher consisting of several rounds (10, 12, or 14, depending on key size). Each round involves the four steps called *SubBytes* (byte substitution, or S-box), *ShiftRows*, *MixColumns*, and *AddRoundKey* (the last round skips *MixColumns*, and there is a Round 0 consisting solely of *AddRoundKey*). The latter three steps are linear with respect to the data block, and provide "diffusion." *SubBytes* is the nonlinear step that provides "confusion."

The S-box, applied to each byte, consists of two substeps: (i) considering the byte an element of the Galois field $GF(2^8)$, find its inverse in that field; (ii) considering the resulting byte a vector of bits in $GF(2^2)$, multiply by a given bit matrix and add a given constant vector, i.e., an affine transformation.

In the particular Galois field of AES, a byte represents a polynomial where the bits are coefficients of corresponding powers of x, and multiplication is modulo the irreducible polynomial $q(x) = x^8 + x^4 + x^3 + x + 1$. Equivalently, one could consider a root, say θ, of this polynomial, so $q(\theta) = 0$ in this field; then the bits of a byte would correspond to coefficients of powers of θ, e.g., $2 = \theta, 3 = \theta + 1, 4 = \theta^2$, etc. Thus the bits form a vector with respect to what is called a polynomial basis. But there are computational advantages to considering a different (though isomorphic) representation of $GF(2^8)$. Instead of a vector of dimension eight over $GF(2)$, we consider a byte as a vector of dimension two over $GF(2^4)$, where each 4-bit element is in turn a vector of dimension two over $GF(2^2)$, and finally each 2-bit element is a vector of dimension two over $GF(2)$. This has been called a composite field, or "tower field" representation[10]. In this way, the 8-bit inverse calculation comprises several 4-bit operations, each consisting of various simple 2-bit calculations. For each of these subfields, we have found that a normal basis (consisting of a conjugate pair) is more efficient than a polynomial basis for the required inverse calculation[2].

Converting between the standard AES representation and the composite field representation amounts to a change of basis, accomplished by multiplying the bit vector by a bit matrix. In converting back, this bit matrix can be combined with that of the affine transformation substep[13]. With regard to an additive mask, these matrix multiplies are simple linear calculations for the mask correction terms. Below we detail the mask corrections required for the nonlinear inverse calculation.

2.1 Inversion without masking

Here we employ the following convention: upper-case bold symbols represent elements of the main field (e.g. $\mathbf{A} \in GF(2^8)$); upper-case italic symbols are for elements of the subfield (e.g. $A \in GF(2^4)$); lower-case bold is used for the sub-subfield (e.g. $\mathbf{a} \in GF(2^2)$); and lower-case italic is for single bits (e.g. $a \in GF(2)$).

Without masking, inversion in $GF(2^8)/GF(2^4)$ using a normal basis $[\mathbf{Y}^{16}, \mathbf{Y}]$, where $\mathbf{Y}$ and $\mathbf{Y}^{16}$ are the roots of $\mathbf{X}^2 + \mathbf{X} + N$ and $N \in GF(2^4)$ is the norm (product: $N = \mathbf{Y}^{17}$), is given by:

$$\begin{aligned} \mathbf{A} &= A_1\,\mathbf{Y}^{16} + A_0\,\mathbf{Y} \qquad \text{(given)} && (1)\\ B &= A_1 \otimes A_0 \;\oplus\; N \otimes (A_1 \oplus A_0)^2 && (2)\\ \mathbf{A}^{-1} &= \left(A_0 \otimes B^{-1}\right) \mathbf{Y}^{16} + \left(A_1 \otimes B^{-1}\right) \mathbf{Y} \qquad \text{(result)} && (3) \end{aligned}$$

(Note that $\otimes$ and $\oplus$ denote multiplication and addition calculations in a Galois field, while $A_1\,\mathbf{Y}^{16} + A_0\,\mathbf{Y}$ is just the algebraic expression for the vector $[A_1, A_0]$ in the normal basis.) This requires inversion, multiplication, and the combined "square-scale" operation in the subfield $GF(2^4)$. Similarly, the inversion in $GF(2^4)/GF(2^2)$ using a normal basis $[Z^4, Z]$, where Z and Z^4 are the roots of $X^2 + X + \mathbf{n}$ and $\mathbf{n} \in GF(2^2)$ is the norm ($\mathbf{n} = Z^5$), is given by:

$$\begin{aligned} B &= \mathbf{b}_1\, Z^4 + \mathbf{b}_0\, Z \qquad \text{(given)} && (4)\\ \mathbf{c} &= \mathbf{b}_1 \otimes \mathbf{b}_0 \;\oplus\; \mathbf{n} \otimes (\mathbf{b}_1 \oplus \mathbf{b}_0)^2 && (5)\\ B^{-1} &= \left(\mathbf{b}_0 \otimes \mathbf{c}^{-1}\right) Z^4 + \left(\mathbf{b}_1 \otimes \mathbf{c}^{-1}\right) Z \qquad \text{(result)} && (6) \end{aligned}$$

But in the sub-subfield $GF(2^2)$, inversion is the same as squaring, equivalent to a bit swap:

$$\begin{aligned} \mathbf{c} &= c_1\,\mathbf{w}^2 + c_0\,\mathbf{w} \qquad \text{(given)} && (7)\\ \mathbf{c}^{-1} &= c_0\,\mathbf{w}^2 + c_1\,\mathbf{w} \qquad \text{(result)} && (8) \end{aligned}$$

where $\mathbf{w}$ and $\mathbf{w}^2$ are the roots of $\mathbf{x}^2 + \mathbf{x} + 1$.

2.2 Masked Inversion

Now introduce additive masking. By adding a "random" mask, such that the statistical distribution of masks appears uniform over the field, now our operands appear random as well, uncorrelated to either plaintext or key. Hence the statistical data available through side channels looks like noise, regardless of the chosen sets of plaintexts, and the key is protected. The cost is the computation of mask correction terms.

We use the insight of Oswald *et al.* that in the sub-subfield $GF(2^2)$ inversion (squaring) is additive, so for data $\mathbf{a}$ and mask $\mathbf{m}$, then

$$(\mathbf{a} \oplus \mathbf{m})^{-1} = (\mathbf{a} \oplus \mathbf{m})^2 = \mathbf{a}^2 \oplus \mathbf{m}^2 = \mathbf{a}^{-1} \oplus \mathbf{m}^{-1} \tag{9}$$

and finding the mask correction $\mathbf{m}^2$ is trivial. Hence the tower-field approach eliminates the need to remove the additive mask (or change it to a multiplicative one) before inversion.

In the larger fields, here is how the mask corrections can be calculated. We indicate the masked version of the input byte $\mathbf{A}$ with a tilde: $\tilde{\mathbf{A}}$, and similarly for the other masked quantities. So the input byte to the masked $GF(2^8)$ inverter is

$$\tilde{\mathbf{A}} = (\mathbf{A} \oplus \mathbf{M}) = \tilde{A}_1\, \mathbf{Y}^{16} + \tilde{A}_0\, \mathbf{Y}\ , \tag{10}$$

being the data byte $\mathbf{A}$ already masked by the (known) mask $\mathbf{M} = M_1\, \mathbf{Y}^{16} + M_0\, \mathbf{Y}$. Let

$$\begin{aligned} \tilde{B} \ &= \ \tilde{A}_1 \otimes \tilde{A}_0 \ \oplus \ N \otimes \left(\tilde{A}_1 \oplus \tilde{A}_0\right)^2 && (11)\\ &\quad \oplus \ \tilde{A}_1 \otimes M_0 \ \oplus \ \tilde{A}_0 \otimes M_1 \ \oplus \ M_1 \otimes M_0 && (12) \end{aligned}$$

where the second line shows the additional correction terms required, and the result $\tilde{B}$ is B above, masked by $N \otimes (M_1 \oplus M_0)^2$. Note that, since M_1 and M_0 are random, then so is their sum, so is its square (an isomorphism), and so is the square scaled by $N \neq 0$ (a bijection), that is, the uniform distribution of masks is preserved.

For the subfield inversion, say $\tilde{B} = \tilde{\mathbf{b}}_1\, Z^4 + \tilde{\mathbf{b}}_0\, Z$, call the mask $M_2 = N \otimes (M_1 \oplus M_0)^2 = \mathbf{m}_1\, Z^4 + \mathbf{m}_0\, Z$, and let

$$\begin{aligned} \tilde{\mathbf{c}} \ &= \ \tilde{\mathbf{b}}_1 \otimes \tilde{\mathbf{b}}_0 \ \oplus \ \mathbf{n} \otimes \left(\tilde{\mathbf{b}}_1 \oplus \tilde{\mathbf{b}}_0\right)^2 && (13)\\ &\quad \oplus \ \tilde{\mathbf{b}}_1 \otimes \mathbf{m}_0 \ \oplus \ \tilde{\mathbf{b}}_0 \otimes \mathbf{m}_1 \ \oplus \ \mathbf{m}_1 \otimes \mathbf{m}_0 && (14) \end{aligned}$$

so $\tilde{\mathbf{c}}$ is $\mathbf{c}$ above, masked by $\mathbf{n} \otimes (\mathbf{m}_1 \oplus \mathbf{m}_0)^2$, and again the uniform distribution of masks is preserved.

In the sub-subfield, say $\tilde{\mathbf{c}} = \tilde{c}_1\, \mathbf{w}^2 + \tilde{c}_0\, \mathbf{w}$, and let

$$\tilde{\mathbf{c}}^{-1} \ = \ \tilde{c}_0\, \mathbf{w}^2 + \tilde{c}_1\, \mathbf{w} \text{ (bit swap)} \tag{15}$$

so $\tilde{\mathbf{c}}^{-1}$ is $\mathbf{c}^{-1}$ above masked by another uniform mask. For later convenience, give this mask a name: $\mathbf{m}_2 = \mathbf{n}^2 \otimes (\mathbf{m}_1 \oplus \mathbf{m}_0)$.

The next steps involve only multiplications, which do *not* preserve the uniform distribution of masking. Hence (as in Oswald *et al.*[9]) we need to introduce another additive mask. This mask could be new, or could be re-used bits from the original mask $\mathbf{M}$. In either case, this mask must be added first, *before* all the other mask correction terms are added, to prevent unmasking the operands.

Say now we introduce a new temporary 4-bit mask $T = \mathbf{t}_1\, Z^4 + \mathbf{t}_0\, Z$, and let

$$\begin{aligned}
\tilde{\mathbf{b}}_1^{-1} &= \tilde{\mathbf{b}}_0 \otimes \tilde{\mathbf{c}}^{-1} && (16)\\
&\quad \oplus\ \mathbf{t}_1\ \oplus\ \tilde{\mathbf{b}}_0 \otimes \mathbf{m}_2\ \oplus\ \mathbf{m}_0 \otimes \tilde{\mathbf{c}}^{-1}\ \oplus\ \mathbf{m}_0 \otimes \mathbf{m}_2 && (17)\\
\tilde{\mathbf{b}}_0^{-1} &= \tilde{\mathbf{b}}_1 \otimes \tilde{\mathbf{c}}^{-1} && (18)\\
&\quad \oplus\ \mathbf{t}_0\ \oplus\ \tilde{\mathbf{b}}_1 \otimes \mathbf{m}_2\ \oplus\ \mathbf{m}_1 \otimes \tilde{\mathbf{c}}^{-1}\ \oplus\ \mathbf{m}_1 \otimes \mathbf{m}_2 && (19)
\end{aligned}$$

so that the result $\tilde{B}^{-1} = \tilde{\mathbf{b}}_1^{-1}\, Z^4 + \tilde{\mathbf{b}}_0^{-1}\, Z$ is B^{-1} above, masked by T (but is *not* the inverse of $\tilde{B}$).

Similarly, introduce a new 8-bit mask $\mathbf{S} = S_1\, \mathbf{Y}^{16} + S_0\, \mathbf{Y}$ for the output, and let

$$\begin{aligned}
\tilde{A}_1^{-1} &= \tilde{A}_0 \otimes \tilde{B}^{-1} && (20)\\
&\quad \oplus\ S_1\ \oplus\ \tilde{A}_0 \otimes T\ \oplus\ M_0 \otimes \tilde{B}^{-1}\ \oplus\ M_0 \otimes T && (21)\\
\tilde{A}_0^{-1} &= \tilde{A}_1 \otimes \tilde{B}^{-1} && (22)\\
&\quad \oplus\ S_0\ \oplus\ \tilde{A}_1 \otimes T\ \oplus\ M_1 \otimes \tilde{B}^{-1}\ \oplus\ M_1 \otimes T && (23)
\end{aligned}$$

so that the result $\tilde{\mathbf{A}}^{-1} = \tilde{A}_1^{-1}\, \mathbf{Y}^{16} + \tilde{A}_0^{-1}\, \mathbf{Y}$ is the answer $\mathbf{A}^{-1}$ above, masked by the output mask $\mathbf{S}$:

$$\tilde{\mathbf{A}}^{-1} = \mathbf{A}^{-1} \oplus \mathbf{S} \tag{24}$$

2.3 Re-using Masks

Oswald *et al.* showed that through using parts of the input mask for the intermediate results and the output, then several operations can be eliminated, notably multiplications. We will follow the same strategy below.

The first place where re-using masks helps is in the masked intermediate result $\tilde{\mathbf{c}}^{-1}$, where for one subsequent calculation the mask $\mathbf{m}_1$ would be helpful but for another the preferred mask would be $\mathbf{m}_0$, so we follow Oswald and switch masks. Then starting at (15) above we modify the calculation as follows:

$$\tag{25}$$

$$\begin{aligned}
\tilde{\mathbf{c}}^{-1} &= \tilde{c}_0\, \mathbf{w}^2 + \tilde{c}_1\, \mathbf{w} && (26)\\
&\quad \oplus\ (\mathbf{m}_1\ \oplus\ \mathbf{m}_2) && (27)\\
\tilde{\mathbf{b}}_1^{-1} &= \tilde{\mathbf{b}}_0 \otimes \tilde{\mathbf{c}}^{-1} && (28)\\
&\quad \oplus\ \mathbf{m}_{11}\ \oplus\ \underline{\tilde{\mathbf{b}}_0 \otimes \mathbf{m}_1}\ \oplus\ \mathbf{m}_0 \otimes \tilde{\mathbf{c}}^{-1}\ \oplus\ \underline{\mathbf{m}_0 \otimes \mathbf{m}_1} && (29)\\
\tilde{\mathbf{c}}^{-1} &= \tilde{\mathbf{c}}^{-1}\ \oplus\ (\mathbf{m}_1 \oplus \mathbf{m}_0) && (30)\\
\tilde{\mathbf{b}}_0^{-1} &= \tilde{\mathbf{b}}_1 \otimes \tilde{\mathbf{c}}^{-1} && (31)\\
&\quad \oplus\ \mathbf{m}_{10}\ \oplus\ \underline{\tilde{\mathbf{b}}_1 \otimes \mathbf{m}_0}\ \oplus\ \mathbf{m}_1 \otimes \tilde{\mathbf{c}}^{-1}\ \oplus\ \underline{\mathbf{m}_1 \otimes \mathbf{m}_0} && (32)
\end{aligned}$$

where the underlined products had already been computed previously and may be re-used. (Parens indicate the order of evaluation necessary to avoid unmasking operands, but those combinations were also available from previous computation.) Note also that the result

$\tilde{B}^{-1} = \tilde{\mathbf{b}}_1^{-1} Z^4 + \tilde{\mathbf{b}}_0^{-1} Z$ is still B^{-1} above, but now masked by $M_1 = \mathbf{m}_{11} Z^4 + \mathbf{m}_{10} Z$, the upper half of the input mask. Following the same approach of switching masks at the next level gives

$$\begin{aligned}
\tilde{A}_1^{-1} &= \tilde{A}_0 \otimes \tilde{B}^{-1} && (33)\\
&\quad \oplus\ S_1 \ \oplus\ \underline{\tilde{A}_0 \otimes M_1} \ \oplus\ M_0 \otimes \tilde{B}^{-1} \ \oplus\ \underline{M_0 \otimes M_1} && (34)\\
\tilde{B}^{-1} &= \tilde{B}^{-1} \oplus (M_1 \ \oplus\ M_0) && (35)\\
\tilde{A}_0^{-1} &= \tilde{A}_1 \otimes \tilde{B}^{-1} && (36)\\
&\quad \oplus\ S_0 \ \oplus\ \underline{\tilde{A}_1 \otimes M_0} \ \oplus\ M_1 \otimes \tilde{B}^{-1} \ \oplus\ \underline{M_1 \otimes M_0} && (37)
\end{aligned}$$

again allowing the underlined terms to be re-used, and with the output $\tilde{\mathbf{A}}^{-1} = \tilde{A}_1^{-1}\, \mathbf{Y}^{16} + \tilde{A}_0^{-1}\, \mathbf{Y}$ being masked by the output mask $\mathbf{S}$ (which could be the original intput mask $\mathbf{M}$, or not):

$$\tilde{\mathbf{A}}^{-1} = \mathbf{A}^{-1} \oplus \mathbf{S} \qquad (38)$$

2.4 Re-using Masks between rounds

Many of the mask correction terms used in the masked inversion above involve *only* the input mask, independent of the masked data. This is also true of *all* the mask correction term calculations in the other steps of each round of encryption, as those other steps are all linear (with respect to the additive mask). Then, if the original 128-bit mask for a block of data were to be re-used for every round, all those data-independent correction terms would be the same for each round. For implementations where the round loop is "unrolled" with S-boxes for each round, these terms would only need computing once, then could be passed along to all the other rounds. This would save the re-computation of all those mask terms, eliminating the associated circuitry, at the modest cost of the "wiring" required to pass along the correction terms. Of course, one would use a new random mask with each new block of data in Round 0, to ensure that over time the distribution of masks remains uniform.

More precisely, one way to do this starts by picking a random 128-bit mask that will be used as the *output* mask (whose bytes correspond to $\mathbf{S}$ above) from the inversion step. Then after each byte undergoes the basis change (from the tower field form) and affine transformation part of the S-box (excluding the additive constant), the ShiftRows step is applied to the whole mask; the result is the output mask after the last round of encryption (which lacks the MixCols step). Then MixCols is applied to that, giving the *input* mask to be added to the initial data before Round 0. Applying byte-wise the basis change (to the tower field form) gives the input mask (corresponding to $\mathbf{M}$ above) for the inversion step. From this can be computed such terms as $M_1 \otimes M_0$, M_2, $\mathbf{m}_1 \otimes \mathbf{m}_0$, and $\mathbf{m}_2$ above, to be re-used each round. Then the only correction terms that would need computing in each round are the data-dependent terms (e.g. $\tilde{A}_1 \otimes M_0$ above) of the inversion step.

But this only makes sense if the application has enough room to unroll the loop. In cases where compactness is paramount the same few S-boxes would be employed for each round; using pre-computed correction terms from round to round would then require extra registers – a cost rather than a saving.

2.5 Security of Masks

Here we show that this masked inversion operation is secure, by which we mean, given input and output masks uniformly distributed, then the distribution of each masked operand is independent of the distribution of the data.

First note that, for a variable $x \in \mathbb{F}$ uniformly distributed in a *finite* field $\mathbb{F}$, then applying any bijection (one-to-one, onto mapping from the field to itself) $f : \mathbb{F} \to \mathbb{F}$ will give a new variable $y = f(x)$ that is also uniformly distributed. In particular, any isomorphism is a bijection.

Also, for a string of n bits $[b_1, b_2, \cdots, b_n]$ uniformly distributed over the set of all 2^n such strings of the same length, then any substring $[b_i, b_{i+1}, \cdots, b_j]$ will also be uniformly distributed over the set of such strings of the same length.

Now consider the operations we perform with the initial, uniformly distributed masks m. Adding data a to give masked data $\tilde{a} = a \oplus m$ is a bijection $f(m) = m \oplus a = \tilde{a}$; regardless of what the data a is, the masked data $\tilde{a}$ retains the uniform distribution of the mask. Splitting a mask into two halves gives two independent masks uniformly distributed over the subfield. Adding two independent masks results in another uniformly distributed mask. Squaring is an isomorphism in $GF(2^n)$, so squaring a mask gives another uniformly distributed mask. Similarly to addition, multiplying by a *nonzero* constant value $n \neq 0$ is a bijection; here the constant is the norm of the basis elements and so cannot be zero. So, for example, M_2 above, the mask for B, retains the uniform distribution of $\mathbf{M}$ from which it was derived using the above operations; similarly $\mathbf{m}_2$.

Multiplying two independent uniformly distributed variables does *not* give a uniformly distributed product. The latter half of the inverse calculation involves only multiplications, including mask correction terms, so no such term can act as a mask, and adding all such terms would unmask the operand. (Note that each of these individual products has the same distribution as the product of two random variables, so is not related to the unmasked data.) Here, *first* starting with a new uniformly distributed mask and *then* adding products to it will ensure that each intermediate result maintains the uniform distribution of the mask, as pointed out by [9]. Therefore, the distribution for each intermediate term (either uniform or product of uniform) is independent of the data, and the calculation is secure.

3 Implementation Details

The appendix gives Verilog code for a masked S-box using the merged architecture of Satoh *et al.*, which combines the S-box with the inverse S-box (for decryption), sharing the Galois inverter. The tower field representation here uses the same normal bases used by Canright in the unmasked version. Here both the input mask and the output mask are parameters, along with the masked data byte. The code has been tested in an FPGA implementation and shown to give correct results for every combination of encryption/decryption, data byte, input mask, and output mask (33,554,432 combinations).

The same types of optimizations for minimal circuitry as used by Canright for the unmasked S-box were applied to the masked version. Among these optimizations are the re-use of bit sums for factors in multipliers (using normal bases, all factors are shared between two

Table 1: Inverter Size. Here we compare the masked inverter with the unmasked version, where total gates is in NAND equivalents.

Inverter	gate counts	total gates
masked	217 XOR, 94 NAND, 6 NOR	480
unmasked	56 XOR, 34 NAND, 6 NOR	138

Table 2: Basis Change Sizes. Here we compare gates needed in the basis change bit matrices (including the affine transformation but excluding the Galois inverter) for a merged S-box & inverse, S-box alone, and inverse S-box alone, using different input and output masks, same mask for both, or no mask. Both individual gate counts and NAND equivalents are given.

Basis Change	merged	S-box	(S-box)$^{-1}$
2 masks	78 XOR, 4 NOT, 32 MUX = 196	49 XOR = 86	50 XOR = 88
1 mask	58 XOR, 3 NOT, 24 MUX = 146	44 XOR = 77	45 XOR = 79
unmasked	38 XOR, 2 NOT, 16 MUX = 96	24 XOR = 42	25 XOR = 44

multipliers) and the use of NOR gates where appropriate to replace a combination of NAND and XOR gates (minimizing the size for the 0.13-μ CMOS standard cell library[14] considered). The tables give the results for the masked Galois inverter and the basis change (bit matrices) separately. Results are shown by number and type of logic operations, and also by total "gates," where the number refers to the equivalent number of NAND gates (rounded to whole numbers), using our standard cell library. We use the equivalencies 1 XOR/XNOR = $\frac{7}{4}$ NAND gates, 1 NOR = 1 NAND gate, 1 NOT = $\frac{3}{4}$ NAND gate, and 1 MUX21I = $\frac{7}{4}$ NAND gates [14].

Note that the additional resources needed to use different masks on input and output are significant for the merged architecture, but not for dedicated encryption (or decryption) only. There is little reason not to use the input mask for the output as well. In this case, the size for the merged architecture where encryption and decryption share an inverter is 626 NAND equivalents, almost three times the size of the unmasked version (234). (For encryption only, not merged, the S-box with a single mask for both input and output is 557 NAND equivalents, compared with 180 for unmasked; again masked is three times larger.)

However, if the current approach were used in an application where the loop of rounds was "unrolled" (requiring enough room for at least 160 S-boxes), the masks could be re-used from round to round, as discussed above. This would require passing along the extra bits of pre-computed corrections between rounds. For one S-box, the total number of mask-term bits would be 43, as compared to 8 bits for an input mask alone (to be used as output mask also, or 16 bits for two different masks). These extra wires would replace 33 XORs and 12 NANDs in the inverter, and all of the mask basis change calculation, e.g. 40 XORs, 16 MUXs, and 2 NOTs for the merged S-box with two different masks, or for an S-box alone with one mask, 20 XORs (no MUXs or NOTs). So re-using masks between rounds would give a masked merged S-box of 506 NAND equivalents, rather than the 626 above. In addition to this saving per S-box (after the first round), the MixCols operation on the mask block would also be eliminated (again, after the first round).

Direct comparison with Oswald *et al.*[9] is difficult at the level of optimization employed here. Their terms of comparison are operations in $GF(2^4)$, excluding addition; in their Table 1 they list 9 multiplications, 2 squarings, and 2 multiplications by a constant (or scalings). In those terms, the present approach is almost the same, except we require only 8 multiplications instead of 9. But each of the two squarings is followed by a scaling (same constant), so our approach treats square-scale as a single operation, which we have optimized down to 3 single-bit XORs, less than one of the 4-bit additions that are not counted there. But while comparison is difficult in the lack of specifics, to the best of our knowledge ours is the smallest masked S-box to date.

4 Conclusion

For some hardware implementations of AES, countermeasures against side-channel attacks can be important. Here we give a method for masking the S-box (the rest of a round being linear) that is secure, in that the distributions of all the masked operands are independent of the distribution of the data. This masked S-box has been optimized for minimal chip area, giving the smallest masked S-box of which we are aware.

The overhead for masking nearly triples the size of the S-box, from 234 gates to 626 gates for the merged version. In applications with sufficient resources to unroll the round loop (where the compactness of the S-box allows more copies for a given area), some savings may result from re-using the block mask between rounds. Then each S-box (after the first round) would require only 506 gates, a little over twice the size of the unmasked version.

Acknowledgements

Many thanks to Lejla Batina for suggesting this topic and other helpful advice.

References

[1] Mehdi-Laurent Akkar and Christophe Giraud. An implementation of DES and AES, secure against some attacks. In *CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–18, 2001.

[2] D. Canright. A very compact S-box for AES. In *CHES2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.

[3] Pawel Chodowiec and Kris Gaj. Very compact FPGA implementation of the AES algorithm. In C.D. Walter et al., editor, *CHES2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2003.

[4] Jovan Dj. Golić and Christophe Tymen. Multiplicative masking and power analysis of AES. In *CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 198–212, 2002.

[5] Kimmo U. Jarvinen, Matti T. Tommiska, and Jorma O. Skytta. A fully pipelined memoryless 17.8 gbps AES128 encryptor. In *FPGA03*. ACM, 2003.

[6] Sumio Morioka and Akashi Satoh. A 10 Gbps full-AES crypto design with a twisted-BDD S-box architecture. In *IEEE International Conference on Computer Design*. IEEE, 2002.

[7] Sumio Morioka and Akashi Satoh. An optimized S-box circuit arthitecture for low power AES design. In *CHES2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 2003.

[8] NIST. Specification for the ADVANCED ENCRYPTION STANDARD (AES). Technical Report FIPS PUB 197, National Institute of Standards and Technology (NIST), November 2001.

[9] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A side-channel analysis resistant description of the AES S-box. In *FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 413–23, 2005.

[10] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, Institute for Experimental Mathematics, University of Essen, Germany, 1994.

[11] Vincent Rijmen. Efficient implementation of the Rijndael S-box. available at `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/sbox.pdf`, 2001.

[12] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient Rijndael encryption implementation with composite field arithmetic. In *CHES2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2001.

[13] A. Satoh, S. Morioka, K. Takano, and Seiji Munetoh. A compact Rijndael hardware architecture with S-box optimization. In *Advances in Cryptology - ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2001.

[14] Akashi Satoh. personal communication, July 2004.

[15] Nicholas Weaver and John Wawrzynek. High performance, compact AES implementations in Xilinx FPGAs. available at `http://www.cs.berkeley.edu/~nweaver/papers/AES_in_FPGAs.pdf`, September 2002.

[16] Johannes Wolkerstorfer, Elisabeth Oswald, and Mario Lamberger. An ASIC implementation of the AES Sboxes. In *CT-RSA*, volume 2271 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 2002.

Appendix: S-box Algorithm in Verilog

```
/* S-box & inverse with MASKING, using all normal bases */
/*   based on compact S-box using Canright algorithm */
/*   optimized using NOR gates and NAND gates */

/* multiply in GF(2^2), shared factors, using normal basis [Omega^2,Omega] */
module GF_MULS_2 ( A, B, Q );
    input  [2:0] A;   /* shared factors include bit sum: sum hi lo */
    input  [2:0] B;
    output [1:0] Q;
    wire   abcd, p, q;

    assign abcd = ~(A[2] & B[2]);  /* note: ~& syntax for NAND won't compile */
    assign p = (~(A[1] & B[1])) ^ abcd;
    assign q = (~(A[0] & B[0])) ^ abcd;
    assign Q = { p, q };
endmodule

/* multiply & scale by N in GF(2^2), shared factors, basis [Omega^2,Omega] */
module GF_MULS_SCL_2 ( A, B, Q );
    input  [2:0] A;   /* shared factors include bit sum: sum hi lo */
    input  [2:0] B;
    output [1:0] Q;
    wire   t, p, q;

    assign t = ~(A[0] & B[0]);  /* note: ~& syntax for NAND won't compile */
    assign p = (~(A[2] & B[2])) ^ t;
    assign q = (~(A[1] & B[1])) ^ t;
    assign Q = { p, q };
endmodule

/* sums for shared factors, 2-bit -> 3 */
module FAC_2 ( a, Q );
    input  [1:0] a;
    output [2:0] Q;
    wire   sa;

    assign sa = a[1] ^ a[0];
/* output is three 1-bit shared factors: sum hi lo */
    assign Q = { sa, a };
endmodule

/* inverse in GF(2^4)/GF(2^2), using normal basis [alpha^8, alpha^2] */
module GF_INV_4 ( A, M, N, O, Q );
```

```
    input  [3:0] A;
    input  [3:0] M;   /* input mask */
    input  [3:0] N;   /* output mask */
    input  [3:0] O;   /* outer mask-switch terms, to save 2 XORs */
    output [3:0] Q;
    wire   [1:0] a, b, m, n, c, e, d, p, q, an, mb, mn, dn, em, pm, qm;
    wire   [2:0] af, bf, mf, nf, ef, df;  /* factors w/ bit sums */

    assign a = A[3:2];
    assign b = A[1:0];
    assign m = M[3:2];
    assign n = M[1:0];
    FAC_2 afac(a, af);
    FAC_2 bfac(b, bf);
    FAC_2 mfac(m, mf);
    assign nf = {O[1],n};
    GF_MULS_2 anmul(af, nf, an);
    GF_MULS_2 mbmul(mf, bf, mb);
    GF_MULS_2 mnmul(mf, nf, mn);
/* optimize section below using NOR gates */
    assign c = {   /* note: ~| syntax for NOR won't compile */
      ~(a[1] | b[1]) ^ (~(af[2] & bf[2])) ,
      ~(af[2] | bf[2]) ^ (~(a[0] & b[0])) }
         ^ an ^ mb ^ mn ;
/* end of NOR optimization */
    assign e = {  /* inverse masked by n (lo input mask) */
      c[0] ^ n[0] ^ mf[2] ,
      c[1] ^ m[1] ^ nf[2] };
    FAC_2 efac(e, ef);
    GF_MULS_2  qmul(ef, af, q);
    GF_MULS_2 emmul(ef, mf, em);
/* NOTE: to maintain masking,
 the output mask N must be added BEFORE p, q are added to other terms */
    assign qm = N[1:0] ^ an ^ em ^ mn; /* mask terms for q (lo output) */
    assign d = {  /* switch masks: n -> m (hi input mask) */
      c[0] ^ O[3] ,
      e[0] ^ m[0] ^ n[0] };
    FAC_2 dfac(d, df);
    GF_MULS_2  pmul(df, bf, p);
    GF_MULS_2 dnmul(df, nf, dn);
    assign pm = N[3:2] ^ mb ^ dn ^ mn; /* mask terms for p (hi output) */
    assign Q = { (pm ^ p), (qm ^ q) };
endmodule

/* multiply in GF(2^4)/GF(2^2), shared factors, basis [alpha^8, alpha^2] */
```

```
module GF_MULS_4 ( A, B, Q );
    input  [8:0] A;   /* shared factors include bit sums: sum hi lo */
    input  [8:0] B;
    output [3:0] Q;
    wire   [1:0] ph, pl, p;

    GF_MULS_SCL_2 summul( A[8:6], B[8:6], p);
    GF_MULS_2 himul(A[5:3], B[5:3], ph);
    GF_MULS_2 lomul(A[2:0], B[2:0], pl);
    assign Q = { (ph ^ p), (pl ^ p) };
endmodule

/* sums for shared factors, 4-bit -> 9 */
module FAC_4 ( a, Q );
    input  [3:0] a;
    output [8:0] Q;
    wire   [1:0] sa;
    wire al, ah, aa;

    assign sa = a[3:2] ^ a[1:0];
    assign al = a[1] ^ a[0];
    assign ah = a[3] ^ a[2];
    assign aa = sa[1] ^ sa[0];
/* output is three 3-bit shared factors: sum hi lo */
    assign Q = { aa, sa, ah, a[3:2], al, a[1:0] };
endmodule

/* inverse in GF(2^8)/GF(2^4), using normal basis [d^16, d] */
module GF_INV_8 ( A, M, N, Q );
    input  [7:0] A;
    input  [7:0] M;   /* input mask */
    input  [7:0] N;   /* output mask */
    output [7:0] Q;
    wire   [3:0] a, b, m, n, o, c, d, e, p, q, m4, an, mb, mn, dn, em, pm, qm;
    wire   [8:0] af, bf, mf, nf, ef, df;  /* factors w/ bit sums */
    wire c1, c2, c3;   /* for temp var */

    assign a = A[7:4];
    assign b = A[3:0];
    assign m = M[7:4];
    assign n = M[3:0];
    assign o = m ^ n;  /* to switch masks below; has useful bits */
    FAC_4 afac(a, af);
    FAC_4 bfac(b, bf);
    FAC_4 mfac(m, mf);
    FAC_4 nfac(n, nf);
    GF_MULS_4 anmul(af, nf, an);
    GF_MULS_4 mbmul(mf, bf, mb);
    GF_MULS_4 mnmul(mf, nf, mn);
/* optimize section below using NOR gates */
    assign c1 = ~(af[5] & bf[5]);
    assign c2 = ~(af[6] & bf[6]);
    assign c3 = ~(af[8] & bf[8]);
    assign c = {   /* note: ~| syntax for NOR won't compile */
      (~(af[6] | bf[6]) ^ (~(a[3] & b[3]))) ^ c1 ^ c3 ,
      (~(af[7] | bf[7]) ^ (~(a[2] & b[2]))) ^ c1 ^ c2 ,
      (~(af[2] | bf[2]) ^ (~(a[1] & b[1]))) ^ c2 ^ c3 ,
      (~(a[0] | b[0]) ^ (~(af[2] & bf[2]))) ^ (~(af[7] & bf[7])) ^ c2 }
         ^ an ^ mb ^ mn ;
/* end of NOR optimization */
    assign m4 = {   /* this is input mask for subfield */
      mf[6] ^ nf[6] ,
      mf[7] ^ nf[7] ,
      mf[2] ^ nf[2] ,
      o[0] };
    GF_INV_4 dinv( c, m4, m, o, d); /* inverse masked by m (hi input mask) */

    FAC_4 dfac(d, df);
    GF_MULS_4  pmul(df, bf, p);
    GF_MULS_4 dnmul(df, nf, dn);
    assign pm = N[7:4] ^ mb ^ dn ^ mn; /* mask terms for p (hi output) */
    assign e = d ^ o;  /* switch masks: m -> n (lo input mask) */
    FAC_4 efac(e, ef);
    GF_MULS_4  qmul(ef, af, q);
    GF_MULS_4 emmul(ef, mf, em);
    assign qm = N[3:0] ^ an ^ em ^ mn; /* mask terms for q (lo output) */
    assign Q = { (pm ^ p), (qm ^ q) };
endmodule

/* S-box basis change with MASKING, using all normal bases */

/* MUX21I is an inverting 2:1 multiplexor */
module MUX21I ( A, B, s, Q );
    input        A;
    input        B;
    input        s;  /* selection switch */
    output       Q;
    assign Q = ~ ( s ? A : B );  /* mock-up for FPGA implementation */
endmodule
```

```
/* select and invert (NOT) byte, using MUX21I */
module SELECT_NOT_8 ( A, B, s, Q );
    input  [7:0] A;
    input  [7:0] B;
    input        s;  /* selection switch */
    output [7:0] Q;
    MUX21I m7(A[7],B[7],s,Q[7]);
    MUX21I m6(A[6],B[6],s,Q[6]);
    MUX21I m5(A[5],B[5],s,Q[5]);
    MUX21I m4(A[4],B[4],s,Q[4]);
    MUX21I m3(A[3],B[3],s,Q[3]);
    MUX21I m2(A[2],B[2],s,Q[2]);
    MUX21I m1(A[1],B[1],s,Q[1]);
    MUX21I m0(A[0],B[0],s,Q[0]);
endmodule

/* find either Sbox or its inverse in GF(2^8), by Canright Algorithm */
/* with MASKING: the input mask M and output mask N must be given */
module bSbox ( A, M, N, encrypt, Q );
    input  [7:0] A;
    input  [7:0] M;
    input  [7:0] N;
    input        encrypt;  /* 1 for Sbox, 0 for inverse Sbox */
    output [7:0] Q;
    wire   [7:0] B, C, D, E, F, G, H, V, W, X, Y, Z;
    wire R1, R2, R3, R4, R5, R6, R7, R8, R9;
    wire S1, S2, S3, S4, S5, S6, S7, S8, S9;
    wire T1, T2, T3, T4, T5, T6, T7, T8, T9;
    wire U1, U2, U3, U4, U5, U6, U7, U8, U9, U10;

/* change basis from GF(2^8) to GF(2^8)/GF(2^4)/GF(2^2) */
/* combine with bit inverse matrix multiply of Sbox */
assign  R1  = A[7] ^ A[5] ;
assign  R2  = A[7] ~^ A[4] ;
assign  R3  = A[6] ^ A[0] ;
assign  R4  = A[5] ~^  R3  ;
assign  R5  = A[4] ^  R4  ;
assign  R6  = A[3] ^ A[0] ;
assign  R7  = A[2] ^  R1  ;
assign  R8  = A[1] ^  R3  ;
assign  R9  = A[3] ^  R8  ;
assign B[7] =  R7  ~^  R8  ;
assign B[6] =  R5  ;
assign B[5] = A[1] ^  R4  ;
assign B[4] =  R1  ~^  R3  ;
assign B[3] = A[1] ^  R2  ^  R6  ;
assign B[2] = ~ A[0] ;
assign B[1] =  R4  ;
assign B[0] = A[2] ~^  R9  ;
assign Y[7] =  R2  ;
assign Y[6] = A[4] ^  R8  ;
assign Y[5] = A[6] ^ A[4] ;
assign Y[4] =  R9  ;
assign Y[3] = A[6] ~^  R2  ;
assign Y[2] =  R7  ;
assign Y[1] = A[4] ^  R6  ;
assign Y[0] = A[1] ^  R5  ;
    SELECT_NOT_8 sel_in( B, Y, encrypt, Z );

// convert masks also, but no additive constant for affine
assign  S1  = M[7] ~^ M[5] ;
assign  S2  = M[7] ~^ M[4] ;
assign  S3  = M[6] ~^ M[0] ;
assign  S4  = M[5]  ^  S3  ;
assign  S5  = M[4]  ^  S4  ;
assign  S6  = M[3]  ^ M[0] ;
assign  S7  = M[2]  ^  S1  ;
assign  S8  = M[1]  ^  S3  ;
assign  S9  = M[3]  ^  S8  ;
assign E[7] =  S7  ~^  S8  ;
assign E[6] =  S5   ;
assign E[5] = M[1]  ^  S4  ;
assign E[4] =  S1  ~^  S3  ;
assign E[3] = M[1]  ^  S2  ^  S6  ;
assign E[2] = ~ M[0] ;
assign E[1] =  S4   ;
assign E[0] = M[2]  ^  S9  ;
assign F[7] =  S2   ;
assign F[6] = M[4]  ^  S8  ;
assign F[5] = M[6] ~^ M[4] ;
assign F[4] =  S9   ;
assign F[3] = M[6]  ^  S2  ;
assign F[2] =  S7   ;
assign F[1] = M[4] ~^  S6  ;
assign F[0] = M[1]  ^  S5  ;
    SELECT_NOT_8 sel_Min( E, F, encrypt, V );

assign  T1  = N[7] ~^ N[5] ;
assign  T2  = N[7] ~^ N[4] ;
assign  T3  = N[6] ~^ N[0] ;
assign  T4  = N[5]  ^  T3  ;
assign  T5  = N[4]  ^  T4  ;
assign  T6  = N[3]  ^ N[0] ;
assign  T7  = N[2]  ^  T1  ;
assign  T8  = N[1]  ^  T3  ;
assign  T9  = N[3]  ^  T8  ;
assign G[7] =  T7  ~^  T8  ;
assign G[6] =  T5   ;
assign G[5] = N[1]  ^  T4  ;
assign G[4] =  T1  ~^  T3  ;
assign G[3] = N[1]  ^  T2  ^  T6  ;
assign G[2] = ~ N[0] ;
assign G[1] =  T4   ;
assign G[0] = N[2]  ^  T9  ;
assign H[7] =  T2   ;
assign H[6] = N[4]  ^  T8  ;
assign H[5] = N[6] ~^ N[4] ;
assign H[4] =  T9   ;
assign H[3] = N[6]  ^  T2  ;
assign H[2] =  T7   ;
assign H[1] = N[4] ~^  T6  ;
assign H[0] = N[1]  ^  T5  ;
    SELECT_NOT_8 sel_Mout( H, G, encrypt, W );

    GF_INV_8 inv( Z, V, W, C );

/* change basis back from GF(2^8)/GF(2^4)/GF(2^2) to GF(2^8) */
/* combine with matrix multiply of Sbox */
assign  U1  = C[7] ^ C[3] ;
assign  U2  = C[6] ^ C[4] ;
assign  U3  = C[6] ^ C[0] ;
assign  U4  = C[5] ~^ C[3] ;
assign  U5  = C[5] ~^  U1  ;
assign  U6  = C[5] ~^ C[1] ;
assign  U7  = C[4] ~^  U6  ;
assign  U8  = C[2] ^  U4  ;
assign  U9  = C[1] ^  U2  ;
assign  U10  =  U3  ^  U5  ;
assign D[7] =  U4  ;
assign D[6] =  U1  ;
assign D[5] =  U3  ;
assign D[4] =  U5  ;
assign D[3] =  U2  ^  U5  ;
assign D[2] =  U3  ^  U8  ;
assign D[1] =  U7  ;
assign D[0] =  U9  ;
assign X[7] = C[4] ~^ C[1] ;
assign X[6] = C[1] ^  U10  ;
assign X[5] = C[2] ^  U10  ;
assign X[4] = C[6] ~^ C[1] ;
assign X[3] =  U8  ^  U9  ;
assign X[2] = C[7] ~^  U7  ;
assign X[1] =  U6  ;
assign X[0] = ~ C[2] ;
    SELECT_NOT_8 sel_out( D, X, encrypt, Q );
endmodule

/* test program: put Sbox output into register */
/* with MASKING: the input mask M and output mask N must be given */
module Sbox_m ( A, M, N, S, Si, CLK );
    input  [7:0] A;
    input  [7:0] M;
    input  [7:0] N;
    output [7:0] S;
    output [7:0] Si;
    input  CLK  /* synthesis syn_noclockbuf=1 */ ;
    reg    [7:0] S;
    reg    [7:0] Si;
    wire   [7:0] s;
    wire   [7:0] si;

    bSbox sbe(A, M, N, 1, s);
    bSbox sbd(A, M, N, 0, si);
    always @ (posedge CLK) begin
    S <= s;
    Si <= si;
    end
endmodule
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library, Code 013
   Naval Postgraduate School
   Monterey, California 93943-5100

3. Professor Clyde L. Scandrett
   Department of Applied Mathematics
   Naval Postgraduate School
   Monterey, California 93943-5216

4. Professor Pante Stanica
   Department of Applied Mathematics
   Naval Postgraduate School
   Monterey, California 93943-5216